# Continuous Assessment – Python Programming

**Time**: 4 hours
**Level**: Level 1 – Software Engineering & Cybersecurity
**Total Points**: 30 pts (+2 Bonus pts)

---

## Part 1: Code Comprehension & Python Concepts (5 pts)

### 1. Code Analysis (3 pts)

```
class User:

    def __init__(self, username, email):

        self.username = username

        self.email = email

    def greet(self):

        return f"Hello {self.username}, your email is
{self.email}"

users = [User("sony", "sony@mail.com"), User("nina",
"nina@mail.com")]

for user in users:

    print(user.greet())
```

**Instructions**:

1. Comment each line of code below to explain what it does:

2. Extend the class `User` by doing the following:

- Add a class variable `user_count` that tracks how many users have been created.
- Modify the constructor to update this variable when a new user is instantiated.
- Add a `classmethod` named `from_string` that takes a string like `"test,test@mail.com"` and returns a `User` object.

- Add a `staticmethod` named *`validate_email`* that checks whether an email contains `"@"` return `True` or `False`.
- Add a `property` method called `contact` that returns the user's full contact as: `username <email>`.
- Explain difference between classmethods, instance methods and static methods
- Test all new methods and print their results.

### 2. List Copying & Immutability (2 pts)

A. What is the output of the following code?

```
a = [1, 2, 3]
b = a
b.append(4)
print(a)
```

B. Fix the behavior to avoid modifying `a` when changing `b`.

C. Explain:

- The difference between **shallow copy** and **deep copy**.
- When and why each is used.

---

## Part 2: Python Concepts & Debugging (5 pts)

### A. Generators (3 pts)

1. What is a generator?
    - Explain how a generator function differs from a regular function under the hood.
    - Describe what a generator does at runtime.
2. How does *`yield`* behave differently from *`return`*?

- Discuss how execution pauses and resumes across `yield` calls.
- Explain what happens when the generator finishes.

3. **Compare a generator function to a generator expression.**
   - Create a generator that *countdown* from *n* passed as parameter to 1. Write the equivalent using list comprehension.
   - Explain one advantage and one drawback of using generators instead of lists.
4. **State Management**

Modify your `countdown` generator so that it accepts a step size via `.send(step)` on each iteration. For example:

```
gen = countdown(10)        # starts at 10
next(gen)                  # yields 10
gen.send(2)                # subtracts 2, yields 8
gen.send(3)                # subtracts 3, yields 5
```

- Briefly explain how the `.send()` method changes the state inside the generator.

## B. Debug the Decorator (2 pts)

```
def log_call(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__}")
        return func(args)
```

```
        return wrapper

    @log_call
    def add(x, y):
        return x + y

    print(add(3, 4))
```

**Instructions**:

- Identify and fix all bugs in the code.
- Explain:
    - What a **decorator** is.
    - How this decorator works.
- Print the correct function result after logging.

---

## Part 3: Mini Project & Debugging (15 pts)

### 3.1 Projects

### Choose TWO projects from the following below the project b is mandatory (10 pts)

#### A. Contact Book with File I/O and Classes

- Create a `Contact` class (name, phone, email)
- Add/remove contacts in a list

- Save and load contacts from a file (e.g., `contacts.json`)
- Use: `__str__`, `@classmethod`, or `@property` as needed

## B. Password Manager

This project involves building a command-line interface (CLI) application that functions as a secure password manager. The application enables users to create an account, manage their credentials, and securely store passwords for various services.

### Key Features

1. **User Registration**
    - Users can register using their email address and a **master password**.
    - Upon registration, a **vault** is created for each user. This vault is represented as a dedicated folder on the file system.
    - The vault folder is uniquely named (e.g., using a UUID or a hashed version of the user's email) and is linked to the user in a JSON file that stores account metadata.
2. **Vault Structure**
    - The vault contains two files to store credentials for different device types:
        - *`websites.txt`*: Stores credentials for online services.
        - *`desktop_software.txt`*: Stores credentials for local software applications.
3. **Saving Credentials**
    - To add a new credential, the user must authenticate with their email and master password.
    - After authentication, the user selects the type of device:

        - **Website**: The user provides the website URL, username, and password.
        - **Desktop Software**: The user provides the software name, username, and password.
    - Credentials are saved to the corresponding file in the user's vault.

4. **Encryption**
    - All stored passwords and username are encrypted using the master password as the encryption key.
    - Decryption is only possible when the correct master password is provided.

5. **User Data Storage**
    - User accounts and vault metadata (e.g., vault folder name) are persisted in a JSON file.

### Security Considerations

- Do not store the master password in plaintext use `hashlib.sha256(b"str").hexdigest()` for hashing
- Use secure file I/O practices to prevent accidental data leakage (python context management).

NB: form encryption use *cryptography* library like following

```
from cryptography.fernet import Fernet

# we will be encrypting the below string.

message = "hello geeks"

# Instance the Fernet class with the key
```

```
        fernet = Fernet(key)

        encMessage = fernet.encrypt(message.encode())
```

### C. Task Manager with Logging

- Create a `Task` class (description, done, priority)
- Add a `@log_action` decorator to log task additions/removals
- Filter tasks by priority or status

## 3.2 Debug the Code (5 pts)

```
    # Welcome to the Python Bug Hunt!

    # Your mission, should you choose to accept it, is to find and
    explain the bugs

    # hidden in this script. This is typical of a Level One university
    Python assignment.

    # Good luck, and may your debugging skills be sharp!


    # --- Configuration Section ---

    DEFAULT_CONFIG = {"version": 1.0, "settings": ["low_power",
    "auto_save"]}

    LOG_FILE_BASENAME = "app_log_"

    MAX_LOG_FILES = 5


    # --- Buggy Function 1: User Profile Management ---

    def add_user_activity(user_id, activity, user_activities={}):

        """

        Adds an activity to a user's activity list.
```

```
    If the user doesn't exist, creates a new entry.
    """
    if user_id not in user_activities:
        user_activities[user_id] = []
    user_activities[user_id].append(activity)
    print(f"Activity '{activity}' added for user '{user_id}'.
Current activities: {user_activities[user_id]}")
    return user_activities


# --- Buggy Function 2: Configuration Cloner ---
def clone_and_modify_config(base_config, new_setting):
    """
    Clones a base configuration and adds a new setting.
    """
    import copy

    # Intention: Create a distinct copy of the configuration
    new_config = copy.copy(base_config) # Hint: Is this copy deep
enough?

    new_config["settings"].append(new_setting)
    new_config["version"] = base_config.get("version", 1.0) + 0.1
    print(f"New config created: {new_config}")
    return new_config
```

```
# --- Buggy Function 3: Log File Processor ---
def process_log_files(log_directory, num_files_to_process):
    """
    Simulates processing a series of log files.
    It should open each file, read a line (simulate processing),
and then move to the next.
    """
    processed_lines = 0
    for i in range(num_files_to_process):
        filename = f"{log_directory}/{LOG_FILE_BASENAME}{i}.txt"
        try:
            # Simulate creating and writing to a log file if it
doesn't exist
            # In a real scenario, these files would pre-exist or
be generated by another process.
            with open(filename, 'a+') as f_check:
                if f_check.tell() == 0: # File is empty
                    f_check.write(f"Log entry for file {i}\n")
                f_check.seek(0) # Go to the beginning to read

            # The "buggy" part: opening the file for processing
            file_handler = open(filename, 'r')
            first_line = file_handler.readline()
            if first_line:
                print(f"Processing '{filename}':
{first_line.strip()}")
```

```
                processed_lines += 1
            # Hint: What happens to file_handler after this?

        except IOError as e:
            print(f"Error accessing file {filename}: {e}")

    print(f"Total lines processed: {processed_lines}")
    return processed_lines

# --- Main Execution ---
def main():
    print("--- Starting Bug Hunt Demonstration ---")

    # Bug 1 Demonstration: Mutable Default Argument
    print("\n--- Testing User Activity Logger ---")
    activities1 = add_user_activity("user123", "login")
    add_user_activity("user123", "view_page")
    activities2 = add_user_activity("user456", "logout") # Problem
might appear here

    if "user123" in activities2 and len(activities2["user123"]) >
1 :
        print("BUG ALERT 1: user123 activities might be
unexpectedly shared or grown!")
```

```
    # Bug 2 Demonstration: Shallow Copy
    print("\n--- Testing Configuration Cloner ---")
    original_config = DEFAULT_CONFIG
    print(f"Original default config: {original_config}")

    config_clone1 = clone_and_modify_config(original_config,
"high_performance")
    print(f"After clone 1, original config: {original_config}") #
Problem here?

    config_clone2 = clone_and_modify_config(original_config,
"dark_mode")
    print(f"After clone 2, original config: {original_config}") #
And here?

    if "high_performance" in original_config["settings"] and
"dark_mode" in original_config["settings"]:
        print("BUG ALERT 2: Original config was unexpectedly
modified!")


    # Bug 3 Demonstration: File Handle Leakage
    print("\n--- Testing Log File Processor ---")
    # Create a dummy log directory for demonstration
    import os
    dummy_log_dir = "temp_log_dir"
    if not os.path.exists(dummy_log_dir):
```

```
        os.makedirs(dummy_log_dir)


    process_log_files(dummy_log_dir, MAX_LOG_FILES)

    print(f"Simulated processing {MAX_LOG_FILES} log files. Check
for resource leaks!")

    # How would you verify a file leak in a real system? (Think
about OS limits)

    print("BUG ALERT 3: Files might not be closed properly in
'process_log_files'.")



    # Cleanup dummy log directory (optional)

    # for i in range(MAX_LOG_FILES):

    #     os.remove(f"{dummy_log_dir}/{LOG_FILE_BASENAME}{i}.txt")

    # os.rmdir(dummy_log_dir)


    print("\n--- Bug Hunt Demonstration Complete ---")


if __name__ == "__main__":

    main()
```

Credit: Google Gemini

**Instructions**:

- This code doesn't works Correct all bugs (Mutability Bug, Shallow Copy Bug, File Memory Leakage)

---

## Part 4: Algorithm (5 pts)

### Group Anagrams

Write a function `group_anagrams(words)` that groups a list of words into sets of anagrams.

- `Input: ["listen", "silent", "enlist", "rat", "tar", "art"]`
- `Output: [["listen", "silent", "enlist"], ["rat", "tar", "art"]]`

**Instructions**:

- Implement the function using sorting or a dictionary.
- Explain your logic using comments.